МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсия

Студент гр. 8304 _____ Завражин Д.Г.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2019

Цель работы

Ознакомиться с основными понятиями и приёмами программной реализации рекурсивных алгоритмов, освоить навыки разработки и написания рекурсивных процедур и функций на языке C++ на примере поставленного задания.

Задание

Вариант 26.

Функция Ф преобразования последовательности α ={a1, a2, ..., an} натуральных чисел в последовательность целых чисел определена следующим образом:

• Ф({})={0};

• Ф({a})={-a};

Если ||α||>1 (т.е. в последовательности больше одного натурального числа), то α = {a} U β = {a} U {b_1 , b_2 , ..., b_m } (U – знак конкатенации). Тогда:

• Ф(α) = {666}, если ∄ i: a ⋮ bi (a ⋮ b означает, что a делится нацело на b);

• Ф(α) = Ф({b_1, b_2, ..., b_{i-1}}) U {999, (a+b_i)} U Ф({b_{i+1}, b_{i+2}, ..., b_m}), где a ⋮ bi и ∄ j: (j<i, a ⋮ bj).

Примеры:

Ф({2, 1})=Ф({}) U {999, (2+1)} U Ф({}) ={0, 999, 3, 0}

Ф({5, 2, 3})={666}

Ф({1, 2})={666}

Ф({12, 5 , 5, 6, 8}) = Ф({5, 5}) U {999, (12+6)} U Ф({8}) = Ф({}) U {999, (5+5)}

U Ф({}) U {999, 18} U {-8} = {0} U {999, 10} U {0} U {999, 18, -8} = {0, 999, 10, 0, 999, 18, -8}

Реализовать функцию Ф рекурсивно.

Структура данных *Sequence*

Для обеспечения представления последовательности была создана структура данных *Sequence* на основе контейнера *std::vector* из стандартной библиотеки языка C++.

Интерфейс структуры данных *Sequence* состоит из:

- Ряда конструкторов;
- Операторов = и [];
- Метода *append*, добавляющей в последовательность элемент, содержимое другой последовательности либо любую их комбинацию;
- Метода *cardinality*, возвращающего количество элементов в последовательности;
- Методов *leftThird* и *rightThird*, возвращающие подпоследовательности согласно требованиям поставленной задачи;
- Метода *represent*, возвращающего строковое представление последовательности.

Реализация структуры данных *Sequence* приведена вместе со всем исходным кодом программы.

Функция *computePhi*

Функция *computePhi* вычисляет значение поставленной в условии задачи функцию Ф от передаваемой ей последовательности, а также выводит сведения о глубине рекурсии, ветвлении в ней и промежуточных результатах вычислений.

Сигнатура функции: Sequence<long long> computePhi(Sequence<long long unsigned> &sequence.

Функция *readFromFile*

Функция *readFromFile* осуществляет построчное чтение из файла с переданным её путём и производит проверку соответствия содержимого файла ожидаемым входным данным. Производятся следующие проверки:

- Проверка на открываемочть файла;
- Проверка на пустоту строки;
- Проверки на отсутствие символов { и };
- Проверка на отсутствие недопустимых символов (в случае их присутствия выводится предупреждение, а сами недопустимые символы игнорируются, что может привести к несколько неожиданным результатам);
- Проверка на натуральность получакмых числовых значений.

Заметим, что функция позволяет разделять числа в представлении входных последовательностей как запятыми, так и просто пробелами.

После чтения из файла каждой строки в случае корректности входных данных функция *readFromFile* выполняет функцию *computePhi* и печатает результат её выполнения в стандартный поток вывода; в случае их некорректности строка пропускается.

Сигнатура функции: void readFromFile(const std::string &filePath).

Функция *main*

Функция main выполняет задачу получения от пользователя пути файла, содержащего входные данные. Это может происходить двумя способами:

1. Посредством передачи в качестве единчтвенного аргумента командной строки;
2. Посредством ввода по прямому запросу программы.

После получения пути программа передаёт его функции readFromFile, выполняющей чтение из файла запускающей функцию *computePhi* на корректных входных данных.

Сигнатура функции: int main(int argc, char* argv[]).

Тестирование программы

Тесты, содержащиеся в файле tests.txt, и важные с точки зрения оценки работ программы фрагменты её вывода приведены в таблице 1.

Таблица 1. Тесты, применяемые при тестирование программы.

№	Тест	Вывод программы
1		Error: line 1 happens to be empty.
2	Test	Error: line 2 does not start with '{'.
3	{	Error: line 3 happens to lack '}'.
4	{45	Error: line 4 happens to lack '}'.
5	{45, 34	Error: line 5 happens to lack '}'.
6	{}	Result: {0}
7	{5}	Result: {-5}
8	{-5}	Warning: line 8 contains an unsupported character '-' which will be ignored. The following sequence was acquired: {5} Result: {-5}
9	{-5.7}	Warning: line 9 contains an unsupported character '-' which will be ignored. Warning: line 9 contains an unsupported character '.' which will be ignored. The following sequence was acquired: {57} Result: {-57}
10	{0}	Error: line 10 contains the number 0, which is not a natural number.
11	{0, 0}	Error: line 11 contains the number 0, which is not a natural number.
12	{2, 1}	Result: {0, 999, 3, 0}
13	{5, 2, 3}	Result: {666}
14	{1, 2}	Result: {666}
15	{12, 5, 5, 6, 8}	Result: {0, 999, 10, 0, 999, 18, -8}
16	{12, 5, 5, 6, 8, 8}	Result: {0, 999, 10, 0, 999, 18, 0, 999, 16, 0}
17	{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}	Result: {0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0, 999, 2, 0}

На всех приведённых выше входных данных программа выдаёт ожидаемый результат; таким образом можно сделать вывод, что данная программа

корректно работает во всех охватываемых составленными тестами случаях.

Вывод

В результате выполнания лабораторной работы была реализована программа, отвечающая всем поставленным условиям и проходящая рассмотренное выше составленное в процессе выполнения работы тестирование. Помимо этого, были на практическом примере отточены навыки проектирования, написания и тестирования рекурсивных алгоритмов, владения языком C++.

Исходный код программы

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>

using std::cin;
using std::cout;
using std::endl;

constexpr size_t INITIAL_SEQUENCE_CAPACITY = 10;
constexpr bool PRINT_RECURSION_INFORMATION = true;


template<class T>
class Sequence
```

```
{
public:
    std::vector<T> vector;

    // overload assignment operator
    Sequence<T> &operator=(const Sequence<T> &sequence)
    {
        this->vector = sequence.vector;
        return *this;
    }

    // overload
    T operator[](size_t index)
    {
        return this->vector.at(index);
    }

    // append the given element(s)
    Sequence<T> &append(T elem)
    {
        this->vector.push_back(elem);
        return *this;
    }

    Sequence<T> &append(const Sequence<T> &elems)
    {
        this->vector.insert(this->vector.end(), elems.vector.begin(), elems.vector.end());
        return *this;
    }
```

```
// parameter packs are used to concisely append to a sequence, as in line 169
template<typename... Ts>
Sequence<T> &append(T elem, Ts...elems)
{
   this->append(elem);
   this->append(elems...);
   return *this;
}

template<typename... Ts>
Sequence<T> &append(const Sequence<T> &elems1, Ts...elems2)
{
   this->append(elems1);
   this->append(elems2...);
   return *this;
}

//overload the costructor to construct a sequence with provided elements
Sequence(){}

Sequence(T elem)
{
   this->append(elem);
}

Sequence(const Sequence<T> &elems)
{
   this->append(elems);
}
```

```
// some convenient member functions

size_t cardinality()
{
   return this->vector.size();
}

Sequence<T> leftThird(size_t pos)
{
   Sequence<T> result;
   // 1 here assures that the sequence-initial element
   // is not considered to be a part of the left segment
   for(size_t i = 1; i < pos && i < this->vector.size(); ++i)
         result.append(this->vector[i]);
   return result;
}

Sequence<T> rightThird(size_t pos)
{
   Sequence<T> result;
   for(size_t i = pos + 1; i < this->vector.size(); ++i)
         result.append(this->vector[i]);
   return result;
}

// print the visual sequence representation
std::string represent()
{
   std::string result = "{";
   if(this->vector.size() > 0)
```

```
        {
            result += std::to_string(this->vector[0]);
            if(this->vector.size() > 1)
                for(auto current = this->vector.begin() + 1, end = this->vector.end();
                    current != end; ++current)
                    result += ", " + std::to_string(*current);
        }
        return  result + "}";
    }
};


Sequence<long long> computePhi(Sequence<long long unsigned> &sequence)
{
    // the depth will be tracked using a static variable
    static size_t depth = 0;
    depth += 1;
    if(PRINT_RECURSION_INFORMATION)
    {
        cout << std::string(depth - 1, ' ') << "|-----------" << endl;
         cout << std::string(depth - 1, ' ') << "| computePhi was called with seq = " <<
sequence.represent() << endl;
        cout << std::string(depth - 1, ' ') << "| depth: " << depth << endl;
         cout << std::string(depth - 1, ' ') << "| cardinality : " << sequence.cardinality()
<< endl;
    }

    Sequence<long long> result;
    // in the case of an empty sequence, {0} shall be returned
    if(sequence.cardinality() == 0)
```

```
{
   if(PRINT_RECURSION_INFORMATION)
      cout << std::string(depth - 1, ' ') << "| empty sequence case" << endl;
   result.append(0);
}
// in the case of a sequence containing a single value,
// a sequence containing only the one opposite to it shall be returned
else if(sequence.cardinality() == 1)
{
   if(PRINT_RECURSION_INFORMATION)
      cout << std::string(depth - 1, ' ') << "| sole element case" << endl;
   result.append(-(long long)sequence[0]);
}
else
{
   // the program shall check whether the first number in the given sequence
   // is divisible by at least one another
   bool isDivisible = false;
   size_t pos = 1;
   for(size_t i = 1; i < sequence.cardinality() && !isDivisible; ++i)
      if(sequence[0] % sequence[i] != 0)
         pos += 1;
      else
         isDivisible = true;
   if(isDivisible)
   {
      if(PRINT_RECURSION_INFORMATION)
      {
         cout << std::string(depth - 1, ' ') << "| tripartite case" << endl;
         cout << std::string(depth - 1, ' ') << "| pos  : " << pos << endl;
```

```
            }
            auto leftThird = sequence.leftThird(pos);
            auto rightThird = sequence.rightThird(pos);
                result.append(computePhi(leftThird), 999, sequence[0] + sequence[pos],
computePhi(rightThird));
        }
        else
        {
            if(PRINT_RECURSION_INFORMATION)
                cout << std::string(depth - 1, ' ') << "| 666 case" << endl;
            result.append(666);
        }
    }
    if(PRINT_RECURSION_INFORMATION)
    {
        cout << std::string(depth - 1, ' ') << "|-----------" << endl;
        cout << std::string(depth - 1, ' ') << "| depth: " << depth << endl;
            cout << std::string(depth - 1, ' ') << "| computePhi exited with value " <<
result.represent() << endl;
    }
    depth -= 1;
    if(depth == 0 && PRINT_RECURSION_INFORMATION)
        cout << "|-----------" << endl;
    return result;
}



void readFromFile(const std::string &filePath)
{
```

```
// used to output debug information
size_t lineCounter = 0;

std::ifstream file(filePath);

// each line in a file is interpreted to contain a sequence
if (file.is_open())
{
   for(std::string line; getline(file, line);)
   {
      cout << endl;
      lineCounter += 1;
      cout << "Line " << lineCounter << endl;
      Sequence<long long unsigned> seq;
      long long unsigned number;

      auto current = line.begin();
      auto end = line.end();
      std::string numberRepresentation = "";

      if(current == end)
      {
         cout << "Error: line " << lineCounter << " happens to be empty." << endl;
         continue;
      }
      if(*current != '{')
      {
         cout << "Error: line " << lineCounter << " does not start with '{'." << endl;
         continue;
      }
```

```
current += 1;

bool curly_bracket_encountered = false;
while(current != end)
{
   if(0x30 <= *current && *current <= 0x39) // i.e. if it is a digit
   {
      numberRepresentation += *current;
   }
   else if(*current == ',')
   {
      if(numberRepresentation.length() > 0)
      {
         std::stringstream(numberRepresentation) >> number;
         if(number == 0)
         {
            cout << "Error: line " << lineCounter
                 << " contains the number 0, which is not a natural number."
                 << endl;
            goto new_line;
         }
         seq.append(number);
         numberRepresentation = "";
      }
   }
   else if(*current == '}')
   {
      if(numberRepresentation.length() > 0)
      {
         std::stringstream(numberRepresentation) >> number;
```

```
            if(number == 0)
            {
                cout << "Error: line " << lineCounter
                    << " contains the number 0, which is not a natural number."
                    << endl;
                goto new_line;
            }
            seq.append(number);
            numberRepresentation = "";
        }
        curly_bracket_encountered = true;
        break;
    }
    else if(*current == ' ' || *current == '\t')
    {
        if(numberRepresentation.length() > 0)
        {
            std::stringstream(numberRepresentation) >> number;
            if(number == 0)
            {
                cout << "Error: line " << lineCounter
                    << " contains the number 0, which is not a natural number."
                    << endl;
                goto new_line;
            }
            seq.append(number);
            numberRepresentation = "";
        }
    }
    else
```

```
            {
                cout << "Warning: line " << lineCounter << " contains an unsupported
character '"
                    << *current << "' which will be ignored." << endl;
            }
            current += 1;
        }
        if(!curly_bracket_encountered)
        {
            cout << "Error: line " << lineCounter << " happens to lack '}'." << endl;
            continue;
        }
        cout << "The following sequence was acquired:" << endl;
        cout << seq.represent() << endl;
        cout << "Result:" << endl;
        cout << computePhi(seq).represent() << endl;
        new_line:
        continue;
      }
   }
   else if(!file.good())
   {
       cout << "The specified path is incorrect or leads to a folder" << endl;
   }
   else
   {
       cout << "Unfortunately, file could not be opened." << endl;
   }
}
```

```
int main(int argc, char* argv[])
{
   std::string filePath = "";
   if(argc == 2)
   {
      filePath = std::string(argv[1]);
      cout << "Specified path:" << endl;
      cout << filePath << endl;
   }
   else if(argc > 2)
   {
      cout << "Too many command line arguments" << endl;
   }
   if(argc != 2)
   {
      cout << "Enter path to a file containing input sequences" << endl;
        cout << "(In is possible to specify it as a sole command line argument)." <<
endl;
       cout << "The file provided with the program is located in \"Test/tests.txt\"" <<
endl;
      cin >> filePath;
   }
   readFromFile(filePath);
   return 0;
}
```